

Computer Science Department

TECHNICAL REPORT

DATA FLOW TESTING IN THE PRESENCE OF
UNEXECUTABLE PATHS

By
Phyllis G. Frankl
Elaine J. Weyuker

Technical Report #208
March 1986

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

NYU COMPSCI TR-208
Frankl, Phyllis G
Data flow testing in the
presence of unexecutable
paths.
c.2



DATA FLOW TESTING IN THE PRESENCE OF
UNEXECUTABLE PATHS

By
Phyllis G. Frankl
Elaine J. Weyuker

Technical Report #208
March 1986

DATA FLOW TESTING IN THE PRESENCE OF UNEXECUTABLE PATHS

Phyllis G. Frankl and Elaine J. Weyuker

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

Abstract

Most test data adequacy criteria based upon path selection have the unfortunate property that for some programs with unexecutable paths, no set of test data is adequate. In this paper we define a new family of adequacy criteria, derived from the data flow testing criteria, which circumvent this problem by only requiring the test data to exercise those definition-use associations which are executable. The inclusion relationship among these criteria is explored.

1. Introduction

Several software test data adequacy criteria are based on the idea that one cannot consider a program to be adequately tested if no test data has caused certain sequences of statements to be executed. These methods generally associate a subset T of the input domain of a program P , with the set \mathcal{P} of paths through P 's flow graph which are executed when the program is run with inputs from T . The test T , or equivalently the set of paths \mathcal{P} , is said to satisfy criterion C for program P (" T is C -adequate for P ") if and only if each of the sequences required by C is a subpath of one of the paths in \mathcal{P} .

The most well-known of these criteria are statement testing, branch testing and path testing which require that the test data cause every node (respectively branch, path) in the program's flow graph to be executed [HOW78,HUA75]. Unfortunately, statement and branch testing can fail to expose many common errors and path testing is usually infeasible since programs with loops may have infinitely many paths [G0075,HOW76]. Several criteria which are based on analysis of the

program's control flow and which are stronger than branch testing but weaker than path testing have been proposed [HOW78,MIL74,WOO80].

Recently, a number of test data adequacy criteria which are based on data flow analysis and which "bridge the gap" between branch testing and path testing have been proposed and studied [RAP82,RAP85,LAS83,NTA84,CLA85]. Tools based on some of them have been implemented [FRA85a,FRA85b,GIR85,KOR85]. These criteria are based on the intuition that one should not feel confident that a variable has been assigned the correct value at some point in the program if no test data causes the execution of a path from the assignment to a point where the variable's value is subsequently used.

All of these criteria suffer from the weakness that for programs with unexecutable paths, it may be impossible for any test set to satisfy the given adequacy criterion. For example, consider a program having a for loop in which the upper bound is always greater than or equal to the lower bound. Such a program has unexecutable paths and therefore cannot be adequately tested using the path testing criterion. Our experience has shown that for many programs, unexecutable paths make it impossible for any test to satisfy a given data-flow testing criterion [FRA85a,FRA85b].

This is clearly an undesirable situation. One property which one would expect a "good" adequacy criterion *C* to have is the *applicability* property: for every program *P* there exists some test which is *C*-adequate for *P* [WEY85]. Not only does the applicability property fail for these criteria, but for each of them it is undecidable for a given program whether test data exists which adequately tests the program.

In this paper we define a new family of adequacy criteria, derived from the data-flow testing criteria proposed in [RAP82,RAP85]. Roughly speaking, for each of these criteria, a test is adequate if and only if it comes "as close as possible" to satisfying the corresponding data flow testing criterion. These criteria will be defined precisely and the relationships between them will be explored in section 3. In section 2 we summarize the theory of data-flow testing, extending it to apply to programs written in Pascal. In section 4 a new data flow testing criterion is introduced and its properties are examined.

2. Data Flow Testing





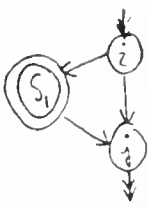
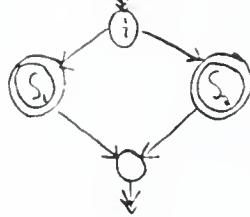
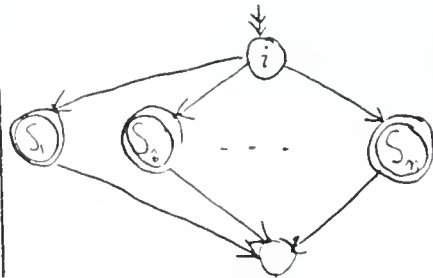
A family of test data selection criteria, each based on the program's data flow characteristics, was defined in [RAP82] and [RAP85] for a very simple universal programming language consisting of assignment statements, conditional and unconditional transfer statements, and input and output statements. These criteria, which we call *data flow testing*, or DF testing for short, require that certain paths from a variable's definition to its subsequent uses be selected. A tool, ASSET, which performs DF testing on programs written in such a language is described in [FRA85a]. We have extended DF testing to apply to Pascal programs and enhanced ASSET accordingly. We now summarize the extended theory of data flow testing.

We apply DF testing to an individual subprogram, i.e., a main program, a procedure, or a function. While treating each array element as an individual variable would in many cases lead to the selection of better test data [FRA85b] doing so is not practical. We therefore treat each array as a single entity. Similarly, occurrences of pointer variables are analyzed purely syntactically; no attempt is made to identify the object to which the pointer points. Each field of a record is treated as an individual variable. Any unqualified occurrence of a record is treated as an occurrence of each field of the record. As a technical convenience we assume that the subprogram does not have **goto** statements, **with** statements, variant records, functions having variable parameters, or procedure or function parameters. We also assume that in every conditional statement at least one variable occurs in the boolean expression which determines the flow of control.

A subprogram can be uniquely decomposed into a set of disjoint *blocks* of statements. A block is a maximal sequence of simple statements having the properties that it can only be entered through the first statement and that whenever the first statement is executed, the remaining statements are executed in the given order. The subprogram to be tested is represented by a *flow graph* in which the nodes correspond to the blocks of the subprogram, and edges indicate possible flow of control between blocks. Figure 1 shows the subgraphs corresponding to statements in the language. The subprogram's flow graph is obtained by merging the exit node of each statement with the entry node of the following statement. An entry node preceding the first statement of the procedure and an exit

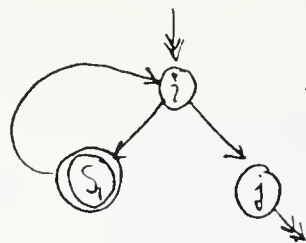
FIGURE 1

(S_1) denotes the subgraph corresponding to S_1 .

<p>assignment statement</p> <p>$v := \text{expr};$</p>		<p>node i has a c-use of each variable in expr followed by a definition of v.</p>
<p>procedure call</p> <p>$P(x_1, \dots, x_n);$</p>		<p>node j has c-uses of x_1, \dots, x_n followed by definitions of x_{i_1}, \dots, x_{i_m} where the parameters in positions i_1, \dots, i_m are variable parameters and the rest are value parameters</p>
<p>input/output statements</p> <p>$\text{read}(x_1, \dots, x_n);$ $\text{read}(f, x_1, \dots, x_n);$ $\text{readln}(x_1, \dots, x_n);$ $\text{readln}(f, x_1, \dots, x_n);$</p>		<p>Node i has definitions of x_1, \dots, x_n. If file variable f is present node i has a c-use of f followed by a definition of f; otherwise node i has a c-use of <i>input</i> followed by a definition of <i>input</i>.</p>
<p>$\text{write}(x_1, \dots, x_n);$ $\text{write}(f, x_1, \dots, x_n);$ $\text{writeln}(x_1, \dots, x_n);$ $\text{writeln}(f, x_1, \dots, x_n);$</p>		<p>Node i has a c-use of x_1, \dots, x_n.</p>
<p>conditional statements</p> <p>if <i>boolean expr</i> then S_1;</p>		<p>Let k be the entry node of S_2</p> <p>Edges (i,j) and (i,k) have p-uses of each variable in <i>boolean expr</i>.</p>
<p>if <i>boolean expr</i> then S_1 else S_2;</p>		<p>Let j,k be the entry nodes of S_1 and S_2</p> <p>Edges (i,j) and (i,k) have p-uses of each variable in <i>boolean expr</i>.</p>
<p>case <i>boolean expr</i> of</p> <p>$l_{11}, \dots, l_{m_1} : S_1;$ $l_{21}, \dots, l_{m_2} : S_2;$</p> <p>$\dots$</p> <p>$l_{n1}, \dots, l_{m_n} : S_n$</p> <p>end;</p>		<p>Let n_1, \dots, n_m be the entry nodes of S_1, \dots, S_m</p> <p>Edges $(i,n_1), \dots, (i,n_m)$ have p-uses of each variable in <i>boolean expr</i>.</p>

repetitive statements

while *boolean expr* do S_1 ;



Let k be the entry node of

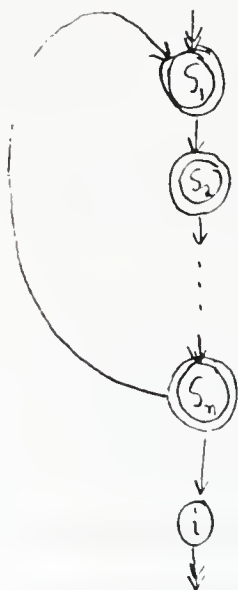
Edges (i,j) and (i,k) have p-uses of each variable in *boolean expr*.

repeat S_1 ;

S_2 ;

S_n

until *boolean expr*;

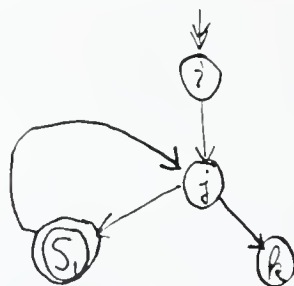


Let j be the entry node of

and let k be the exit node of

Edges (k,j) and (k,i) have p-uses of each variable in *boolean expr*.

for $v := \text{expr1}$ to expr2 do S_1 ;



Let g and h be the entry and exit nodes respectively of

Node i has c-uses of all variables in *expr1* followed by a definition of v . Edges (j,g) and (j,k) have p-uses of v and of all variables in *expr2*. Node h has a c-use of v followed by a definition of v .

FIGURE 1 (cont.)

node succeeding the last statement are added.

Data flow analysis was originally used for compiler optimization, and generally classifies each variable occurrence as being either a *definition*, an *undefinition* or a *use* [HEC77,SCH73]. In addition, we distinguish between two substantially different types of uses. The first type directly affects the computation being performed or allows one to see the result of some earlier definition. We call such a use a computation use or a *c-use*. Of course, a *c-use* may indirectly affect the flow of control through the subprogram. In contrast, the second type of use directly affects the flow of control through the subprogram, and thereby may indirectly affect the computations performed. We call such a use a predicate use or *p-use*. Figure 1 shows the classification of variable occurrences in the language's statements.

We are interested in tracing the flow of data *between* nodes, and thus define a *c-use* of a variable x in node i to be a *global c-use* if the value of x has been assigned in some block other than block i . Let x be a variable occurring in a subprogram. A path (i, n_1, \dots, n_m, j) , $m \geq 0$, containing no definitions or undefinitions of x in nodes n_1, \dots, n_m is called a *definition clear path* with respect to x [def-clear path wrt x] from node i to node j and from node i to edge (n_m, j) . A node i has a *global definition* of a variable x if it has a definition of x and there is a def-clear path wrt x from node i to some node containing a global *c-use* or edge containing a *p-use* of x . Since every *p-use* is associated with a potential transfer of control from one node to another, there is no need to distinguish between *p-uses* and global *p-uses*. The subprogram's *def-use graph* is obtained by associating with each node i , the sets $c\text{-use}(i)$ and $\text{def}(i)$ defined in Figure 2, and with each edge (i, j) the set $p\text{-use}(i, j)$. In addition, the entry node is considered to have a global definition of each parameter, each non-local variable which occurs in the subprogram and the text variable *input* which may be implicitly used in *read* or *readln* statements. The exit node has an *undefinition* of each local variable and a *c-use* of each variable parameter.

A *definition-c-use association* is a triple (i, j, x) where i is a node containing a global definition of x and $j \in \text{dcu}(x, i)$. A *definition-p-use association* is a triple $(i, (j, k), x)$ where i is a node containing a global definition of x and $(j, k) \in \text{dpu}(x, i)$. A *simple path* is one in which all nodes, except possibly

V	= the set of variables
N	= the set of nodes
E	= the set of edges
$\text{def}(i)$	= $\{x \in V \mid x \text{ has a global definition in block } i\}$
$\text{c-use}(i)$	= $\{x \in V \mid x \text{ has a global c-use in block } i\}$
$\text{p-use}(i,j)$	= $\{x \in V \mid x \text{ has a p-use in edge } (i,j) \}$
$\text{dcu}(x,i)$	= $\{j \in N \mid x \in \text{c-use}(j) \text{ and there is a def-clear path from } i \text{ to } j\}$
$\text{dpu}(x,i)$	= $\{(j,k) \in E \mid x \in \text{p-use}(j,k) \text{ and there is a def-clear path from } i \text{ to } (j,k) \}$

Figure 2

the first and last, are distinct. A *loop-free path* is one in which all nodes are distinct. A path (n_1, \dots, n_j, n_k) is a *du-path* with respect to a variable x if n_1 has a global definition of x and either

- i) n_k has a c-use of x and (n_1, \dots, n_j, n_k) is a def-clear simple path with respect to x , or
- ii) (n_j, n_k) has a p-use of x and (n_1, \dots, n_j) is a def-clear loop-free path with respect to x .

An *association* is a definition-c-use association, a definition-p-use association, or a du-path.

A path π *covers* a definition-c-use association (i,j,x) [respectively a definition-p-use association $(i,(j,k),x)$] if it has a definition-clear subpath with respect to x from i to j [respectively, from i to (j,k)]. π covers a du-path π' if π' is a subpath of π . A set \mathcal{P} of paths covers an association if some element of the set does.

A path through a subprogram's flow graph (which we shall refer to in the sequel as a path through the subprogram) is a *control path* if its first node is the entry node and its last node is the exit node. A path is *executable* or *feasible* if there exists some assignment of values to input variables, non-local variables, and parameters which causes the path to be executed. According to this definition, the question of whether or not a given path through a subprogram is executable is independent of the context in which that subprogram is called. However, it may depend on the effects of any procedures or functions which are called along the path. Note that whether or not a particular path is executable depends on the actual subprogram, not just on its def-use graph. Since the sets of paths to which we are applying the criteria arise from the execution of test data we will

always assume that they are sets of executable control paths.

Roughly speaking, the family of data flow testing criteria is based on requiring that the test data execute definition-clear paths from each node containing a global definition of a variable to specified nodes containing global c-uses and edges containing p-uses of that variable. For each variable definition we can demand that $\left[\begin{smallmatrix} all \\ some \end{smallmatrix} \right]$ definition clear paths with respect to that variable from the node containing the definition to $\left[\begin{smallmatrix} all \\ some \end{smallmatrix} \right]$ of the $\left[\begin{smallmatrix} uses \\ c-uses \\ p-uses \end{smallmatrix} \right]$ reachable by some such path be executed.

The criteria are defined precisely in Figure 3.

THE DATA FLOW TESTING CRITERIA	
Test T satisfies criterion C for subprogram P if for each node i and each $x \in \text{def}(i)$ the set \mathcal{P} of paths executed by T covers the following associations:	
CRITERION	ASSOCIATIONS REQUIRED
All-defs	Some (i,j,x) s.t. $j \in \text{dcu}(x,i)$ or some $(i,(j,k),x)$ s.t. $(j,k) \in \text{dpu}(x,i)$.
All-p-uses	All $(i,(j,k),x)$ s.t. $(j,k) \in \text{dpu}(x,i)$.
All-p-uses/some-c-uses	All $(i,(j,k),x)$ s.t. $(j,k) \in \text{dpu}(x,i)$. In addition, if $\text{dpu}(x,i) = \emptyset$ then some (i,j,x) s.t. $j \in \text{dcu}(x,i)$.
All-c-uses/some-p-uses	All (i,j,x) s.t. $j \in \text{dcu}(x,i)$. In addition, if $\text{dcu}(x,i) = \emptyset$ then some $(i,(j,k),x)$ s.t. $(j,k) \in \text{dpu}(x,i)$.
All-uses	All (i,j,x) s.t. $j \in \text{dcu}(x,i)$ and all $(i,(j,k),x)$ s.t. $(j,k) \in \text{dpu}(x,i)$.
All-du-paths	All du-paths from i to j with respect to x for each $j \in \text{dcu}(x,i)$ and all du-paths from i to (j,k) with respect to x for each $(j,k) \in \text{dpu}(x,i)$.
For comparison we also define the criteria all-nodes (respectively all-edges, all-paths) which require that \mathcal{P} cover every node (respectively every edge, every path) in the flow graph.	

Figure 3

Criterion C_1 includes criterion C_2 if and only if for every subprogram, any test which satisfies C_1 also satisfies C_2 . Criterion C_1 strictly includes criterion C_2 , denoted $C_1 \Rightarrow C_2$, if and only if C_1 includes C_2 and for some subprogram P there is a test which satisfies C_2 but does not satisfy C_1 . The notion of *subsumption* in [CLA85] is identical to our notion of inclusion.

Rapps and Weyuker proved that for the simple language for which DF testing was originally defined, subject to certain minor syntactic restrictions, the relationship among the criteria is as shown in Figure 4 [RAP82,RAP85]. Clarke et. al. [CLA85] have shown the relationship of the criteria defined by Laski and Korel [LAS83] and Ntafos [NTA84] to the DF criteria. We have extended the theory of DF testing in such a way that these relations are preserved. Doing so required the inclusion of definitions of all non-local variables in the entry node of the procedure and careful treatment of implicit uses of the text variable *input*.

3. The Feasible Data Flow Testing Criteria

Given a subprogram P and a DF criterion C it may be the case that no test data for P exists which satisfies C . This occurs when none of the paths which cover some association required by C are executable. In such a case, P cannot be adequately tested according to C . In this section we introduce a new family of criteria which are derived from the DF criteria and which circumvent this problem and investigate some of its properties.

We will say that an association is *executable* if there is some executable path which covers it; it is *unexecutable* otherwise. We define subsets $fdcu(x,i) \subseteq dcu(x,i)$ and $fdpu(x,i) \subseteq dpu(x,i)$ consisting only of those associations which are executable. For each DF criterion C we define a new criterion C^* by selecting the required associations from $fdcu(x,i)$ and $fdpu(x,i)$ instead of from $dcu(x,i)$ and $dpu(x,i)$. Precise definitions of these criteria are given in Figure 5. We will refer to the criteria $\{(all-du-paths)^*, (all-uses)^*, (all-p-uses/some-c-uses)^*, (all-c-uses/some-p-uses)^*, (all-p-uses)^*\}$ as *feasible data flow testing criteria*, or FDF criteria, for short.

The FDF criteria satisfy the applicability property: For any subprogram P and any FDF criterion C^* , there is some (possibly empty) test T which satisfies C^* . However, the question of whether a particular T satisfies C for subprogram P is undecidable. In going from the family DF to the family FDF, we have traded the undecidability of the existence question, "is there any test which is C -adequate for P ?" for the undecidability of the recognition problem "is a given test C -adequate for P ?"

Observe that for any DF criterion C , $C \Rightarrow C^*$. We now investigate the inclusion relations

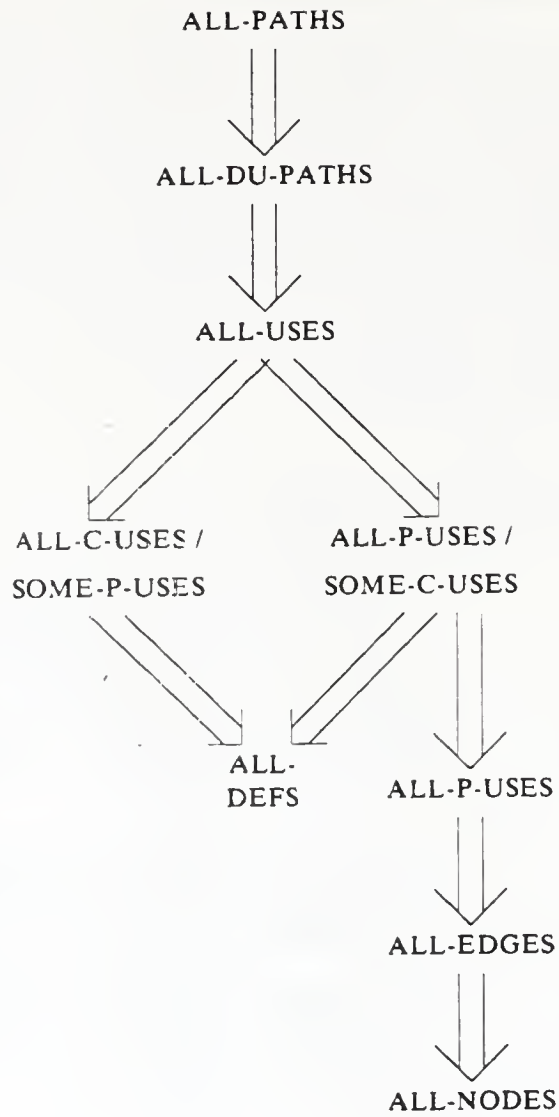


Figure 4

THE FEASIBLE DATA FLOW TESTING CRITERIA

$fdcu(x,i) = \{j \in dcu(x,i) \mid \text{the association } (i,j,x) \text{ is executable}\}$

$fdpu(x,i) = \{(j,k) \in dpu(x,i) \mid \text{the association } (i,(j,k),x) \text{ is executable}\}$

Test T satisfies criterion C for subprogram P if for each node i and each $x \in \text{def}(i)$ the set \mathcal{P} of paths executed by T covers the following associations:

CRITERION	REQUIRED ASSOCIATIONS
(all-defs)*	if $fdcu(x,i) \cup fdpu(x,i) \neq \emptyset$ then some (i,j,x) s.t. $j \in fdcu(x,i)$ or some $(i,(j,k),x)$ s.t. $(j,k) \in fdpu(x,i)$.
(all-p-uses)*	all $(i,(j,k),x)$ s.t. $(j,k) \in fdpu(x,i)$.
(all-p-uses/some-c-uses)*	all $(i,(j,k),x)$ s.t. $(j,k) \in fdpu(x,i)$. In addition, if $fdpu(x,i) = \emptyset$ and $fdcu(x,i) \neq \emptyset$ then some (i,j,x) s.t. $j \in fdcu(x,i)$.
(all-c-uses/some-p-uses/)*	all (i,j,k) s.t. $j \in fdcu(x,i)$. In addition, if $fdcu(x,i) = \emptyset$ and $fdpu(x,i) \neq \emptyset$ then some $(i,(j,k),x)$ s.t. $(j,k) \in fdpu(x,i)$.
(all-uses)*	all (i,j,x) s.t. $j \in fdcu(x,i)$ and all $(i,(j,k),x)$ s.t. $(j,k) \in fdpu(x,i)$.
(all-du-paths)*	all executable du-paths with respect to x from i to j s.t. $j \in dcu(x,i)$ and all executable du-paths with respect to x from i to (j,k) for each $(j,k) \in dpu(x,i)$.

For comparison we also define the criteria (all-nodes)* [respectively (all-edges)*, (all-paths)*] which require that \mathcal{P} cover each executable node [respectively each executable edge, each executable path.]

Figure 5

among the FDF criteria.

THEOREM 1: The family of FDF criteria is partially ordered by strict inclusion as shown in Figure

6. Furthermore, FDF criterion C_i^* includes FDF criterion C_j^* iff and only if it is explicitly shown to do so in the figure or it follows from the transitivity of the relations.

PROOF:

A. Strictness of the inclusions

We first observe that if subprogram P has no unexecutable paths then a test is C-adequate for P if and only if it is C*-adequate for P. This observation, along with the proofs of strictness of the

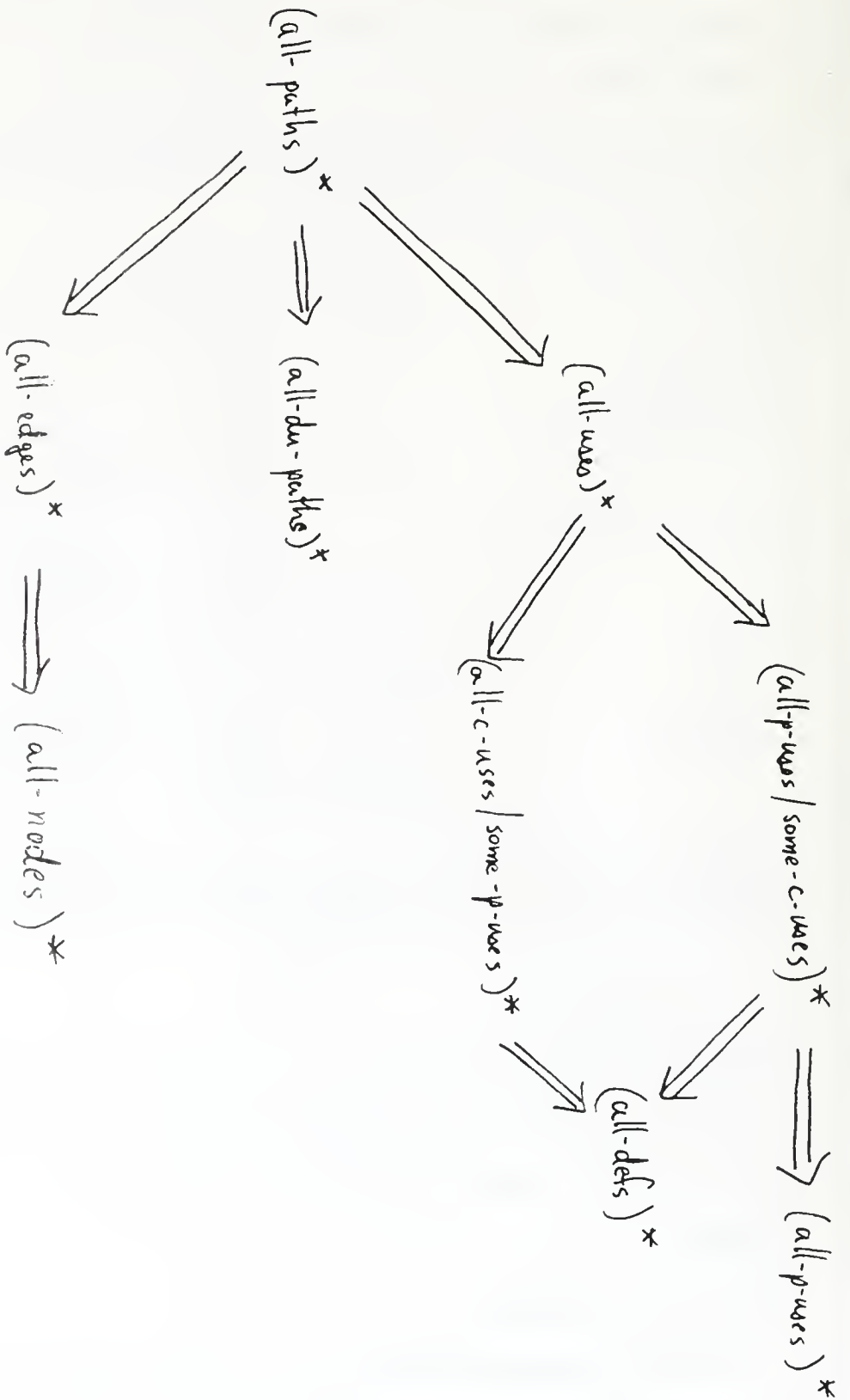


FIGURE 6

inclusions in Theorem 1 of [RAP85], none of which involve subprograms with unexecutable paths, shows that all of the inclusions in Figure 6 are strict. It thus suffices to show that the inclusions in Figure 6 hold.

B.1. $(\text{all-paths})^* \Rightarrow (\text{all-uses})^*$:

Suppose not. Then there is a subprogram P and a set T of test data which is $(\text{all-paths})^*$ -adequate for P but not $(\text{all-uses})^*$ -adequate. Let \mathcal{P} be the set of paths through P which T executes. There exist a node i in P with a global definition of some variable x , a node j with a global c-use of x or edge (j,k) with p-use of x , and an *executable* definition clear path with respect to x from i to j [respectively from i to (j,k)] which is not covered by \mathcal{P} . This contradicts the fact that \mathcal{P} covers every executable path.

The proofs that $(\text{all-paths})^* \Rightarrow (\text{all-du-paths})^*$ and $(\text{all-paths})^* \Rightarrow (\text{all-edges})^*$ are similar and will be omitted.

B.2. $(\text{all-edges})^* \Rightarrow (\text{all-nodes})^*$:

Let T be a test which satisfies $(\text{all-edges})^*$ for subprogram P , and let \mathcal{P} be the set of paths executed by T . Let n be any executable node in P . If n is the entry node, then n has a unique successor, m , and (n,m) is executable. So \mathcal{P} covers (n,m) and hence covers n . If n is not the entry node, then since n is executable, some branch (i,n) is executable. So \mathcal{P} covers (i,n) and hence covers n .

B.3. $(\text{all-uses})^* \Rightarrow (\text{all-p-uses/some-c-uses})^*$, $(\text{all-p-uses/some-c-uses})^* \Rightarrow (\text{all-p-uses})^*$, $(\text{all-p-uses/some-c-uses})^* \Rightarrow (\text{all-defs})^*$, $(\text{all-uses})^* \Rightarrow (\text{all-c-uses/some-p-uses})^*$, $(\text{all-c-uses/some-p-uses})^* \Rightarrow (\text{all-defs})^*$:

These inclusions follow immediately from the definitions of the criteria given in Figure 5. For example, any set \mathcal{P} of paths which covers all of the associations required by $(\text{all-uses})^*$ will *a fortiori* cover all of the associations required by $(\text{all-p-uses/some-c-uses})^*$.

We next show that those relations not in the transitive closure of the diagram in Figure 6 do not hold.

C.1. $(\text{all-du-paths})^* \not\Rightarrow (\text{all-p-uses})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-p-uses/some-c-uses})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-uses})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-c-uses/some-p-uses})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-defs})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-edges})^*$; $(\text{all-du-paths})^* \not\Rightarrow (\text{all-nodes})^*$:

It suffices to show that $(\text{all-du-paths})^* \not\Rightarrow (\text{all-p-uses})^*$, $(\text{all-du-paths})^* \not\Rightarrow (\text{all-defs})^*$, and $(\text{all-du-paths})^* \not\Rightarrow (\text{all-nodes})^*$. The rest follows from the transitivity of \Rightarrow . Consider the subprogram shown in Figure 7a. Its du-paths are shown in Figure 7b. Of these, only (1,2), (2,3,4), (4,3,4), and (4,3,5) are executable. Let $T = \{(X,Y)\}$ where X is any integer and $Y < 0$. Since T executes $\mathcal{P} = \{(1,2,3,4,3,4,3,5,6,8)\}$, T satisfies $(\text{all-du-paths})^*$. However, \mathcal{P} does not cover the associations (2,(5,7),y), (2,7,x), or node 7, all of which are covered by the executable path (1,2,3,4,3,4,3,5,7), so T does not satisfy $(\text{all-p-uses})^*$, $(\text{all-defs})^*$, or $(\text{all-nodes})^*$.

Intuitively, $(\text{all-du-paths})^*$ fails to include these criteria because it is possible for a subprogram to have certain definition-use associations which can be executed only by paths which traverse some loop one or more times. In section 4 we shall introduce a modified version of the $(\text{all-du-paths})^*$ criterion which includes $(\text{all-uses})^*$ and $(\text{all-edges})^*$, and hence all of the other FDF criteria.

C.2. $(\text{all-p-uses})^* \not\Rightarrow (\text{all-edges})^*$; $(\text{all-p-uses/some-c-uses})^* \not\Rightarrow (\text{all-edges})^*$; $(\text{all-uses})^* \not\Rightarrow (\text{all-edges})^*$; $(\text{all-p-uses})^* \not\Rightarrow (\text{all-nodes})^*$; $(\text{all-p-uses/some-c-uses})^* \not\Rightarrow (\text{all-nodes})^*$; $(\text{all-uses})^* \not\Rightarrow (\text{all-nodes})^*$:

Consider the subprogram in Figure 8. Notice that since node 3 is unexecutable, y is always uninitialized at node 4. We will assume that under these circumstances, edges (4,5) and (4,6) are both executable. This would be the case, for example, in an environment in which uninitialized variables receive arbitrary values. Since node 3 is unexecutable, the only executable definition-use associations are (1,2,input), and (2, (2,4), x). Let T be a test which executes $\mathcal{P} = \{(1,2,4,5,7,8)\}$. Then T satisfies $(\text{all-p-uses})^*$, $(\text{all-p-uses/some-c-uses})^*$, and $(\text{all-uses})^*$, but does not satisfy $(\text{all-edges})^*$ or $(\text{all-nodes})^*$.

Notice that the subprogram in Figure 8 has a data flow anomaly [OST76]. We shall see below that this is not a mere coincidence, but that rather, it is this particular kind of anomaly which prevents the inclusions from holding.

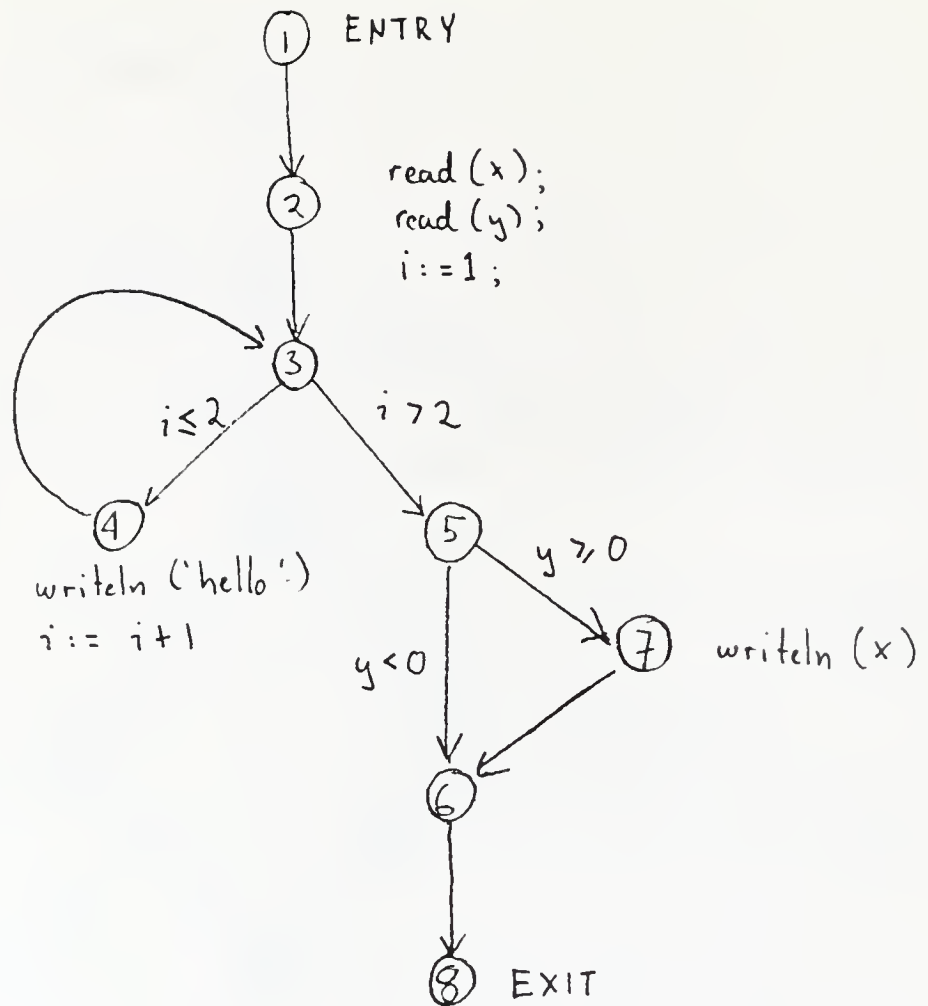


FIGURE 7a.

Path	With respect to	executable
(1,2)	input	yes
(2,3,4)	i	yes
(2,3,5)	i	no
(4,3,4)	i	yes
(4,3,5)	i	yes
(2,3,5,7)	x	no
(2,3,5,6)	y	no
(2,3,5,7)	y	no

Figure 7b

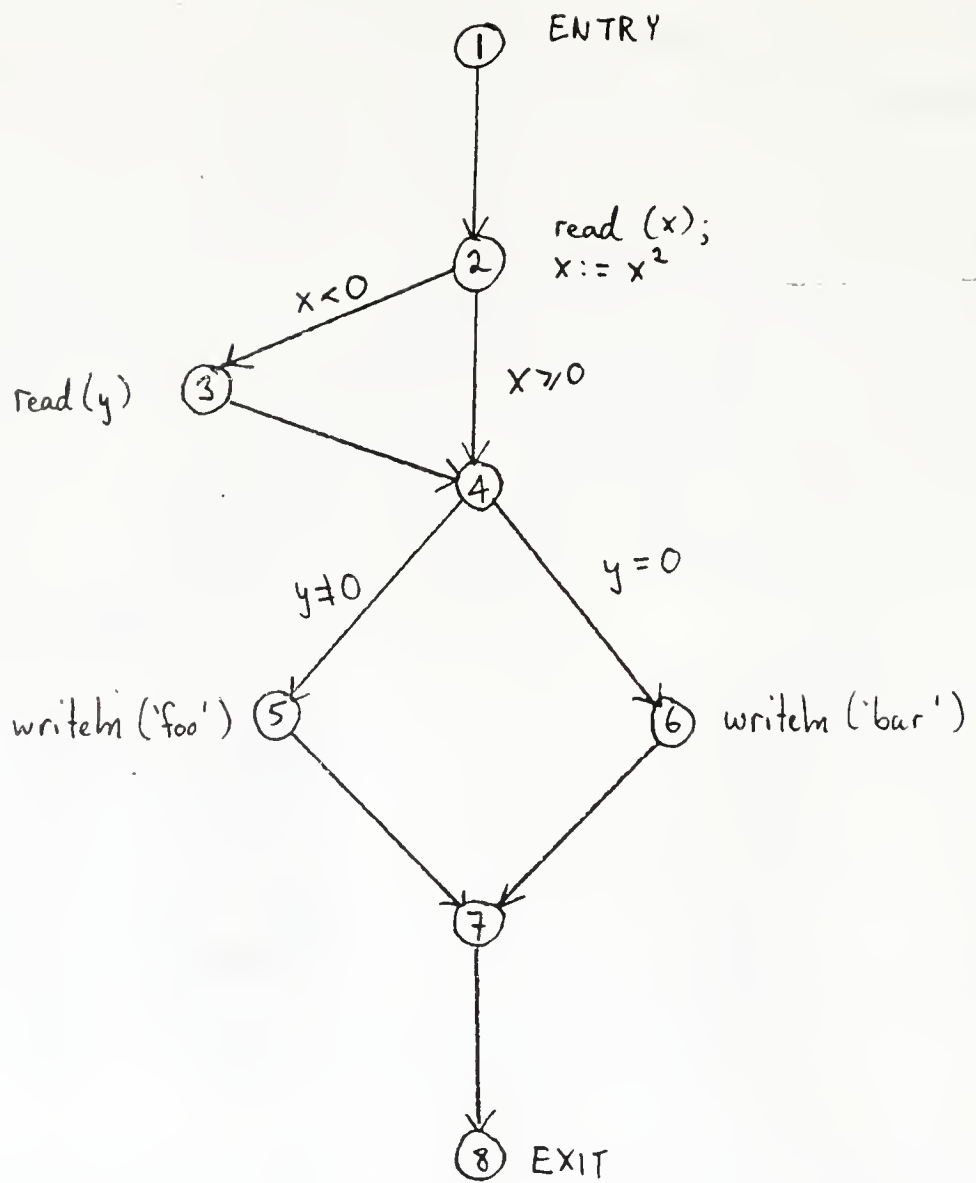


FIGURE 8

The rest of the non-inclusions follow from the incomparability and strictness proofs in [RAP85]. ■

It seems discouraging that $(\text{all-p-uses})^*$ fails to include $(\text{all-edges})^*$. Data flow testing was developed in order to "bridge the gap" between branch testing and path testing. Since many "real-life" subprograms cannot be adequately tested using the unstarred versions of the data flow criteria, one would hope that the FDF criteria would "bridge the gap" between $(\text{all-edges})^*$ and $(\text{all-paths})^*$. We have seen that this is not the case. We next show that for a certain class of "well-behaved" subprograms, any test which satisfies $(\text{all-p-uses})^*$ satisfies $(\text{all-edges})^*$.

DEFINITION: We will say that a subprogram P satisfies the No-Feasible-Undefined-P-uses property (NFUP) if and only if for every executable edge (i,j) in P having a p-use of a variable x there is some *executable* path from the start node to edge (i,j) which contains a global definition of x .

We note that it is quite reasonable to expect subprograms to have property NFUP. If (i,j) is an edge which causes NFUP to fail, then any input which causes (i,j) to be executed will involve referencing an uninitialized variable.

THEOREM 2: For the class of subprograms which satisfy NFUP, $(\text{all-p-uses})^* \Rightarrow (\text{all-edges})^*$.

PROOF: Let P be a subprogram satisfying NFUP, let T be a test which satisfies $(\text{all-p-uses})^*$ for P , let \mathcal{P} be the set of paths executed by T , and let (i,j) be an executable edge in P . Suppose (i,j) has a p-use of a variable x . By hypothesis there is an executable path π from the start node to (i,j) which includes a global definition of x . Let n be the last node in π having a global definition of x . Then $(n, (i,j), x)$ is an executable definition-p-use association so it is covered by \mathcal{P} . Hence (i,j) is covered by \mathcal{P} .

If (i,j) has no p-uses, then the result follows by a straight-forward modification of the corresponding part of the proof of $(\text{all-p-uses}) \Rightarrow (\text{all-edges})$ [RAP85]. ■

In [RAP85] the class of subprograms to which data-flow testing applies is restricted to those subprograms satisfying the No-Syntactic-Undefined-P-use Property (NSUP):

For every p-use of a variable x on an edge (i,j) in P , there is some path from the start node to edge (i,j) which contains a global definition of x .

This restriction was necessary in order to insure that $\text{all-p-uses} \Rightarrow \text{all-edges}$. NFUP is a strengthening of NSUP which takes into account the fact that in subprograms satisfying NSUP, the only paths π from the entry node to some p-use of x such that x has a global definition in some node in π might be unexecutable.

It is tempting to restrict the class of programs to which the FDF criteria apply to those satisfying NFUP. It is our feeling however that while one can live with the undecidability of the adequate test *recognition* problem and perhaps (albeit very uncomfortably) with the undecidability of the adequate test *existence* problem, one should at least be able to decide algorithmically whether a given testing strategy *applies to a given subprogram*. Since it is undecidable whether a given subprogram satisfies NFUP, we refrain from requiring this property of subprograms to be tested.

Another possible way to force $(\text{all-p-uses})^*$ to include $(\text{all-edges})^*$ would be to require subprograms to satisfy the No-Anomalies Property (NA):

Every path from the start node to a use of a variable x must contain a definition of x .

Osterweil and Fosdick [OST76] consider any subprogram not satisfying this property to have data-flow anomaly indicative of possible subprogram error. Since NA is a purely syntactic property and NA implies NFUP we could restrict FDF testing to subprograms satisfying this property. We feel that this is overly restrictive, since many perfectly good subprograms fail to satisfy NA.

One way to force NA to be satisfied is to give the entry node a definition of each variable. Another approach is to perform FDF testing in conjunction with a check for data-flow anomalies. For any subprogram which satisfies NA and any test T which satisfies $(\text{all-p-uses})^*$ the user will be assured that T satisfies $(\text{all-edges})^*$. In case NA does not hold, the user should explicitly check whether $(\text{all-edges})^*$ is satisfied and if necessary add more test data or inspect the subprogram for references of uninitialized variables.

4. A Modification of the All-du-paths Criterion

We have seen above that $(\text{all-du-paths})^*$ fails to include $(\text{all-uses})^*$. This is because in some programs the only *executable* def-clear paths with respect to a variable x from some definition of x to

some use of x are paths with non-simple cycles. In this section we define a modification of the all-du-paths criterion which includes all-uses and whose starred version includes (all-uses)*.

DEFINITION: Let $\pi_x = (n_1, n_2, \dots, n_i)$ be a du-path with respect to x . Then $\text{cycle-extension}(\pi, x) = \{\text{def-clear paths with respect to } x \text{ of the form } (\lambda_1, \lambda_2, \dots, \lambda_k) \text{ where each } \lambda_i \text{ is a path of length greater than or equal to one, beginning and ending with } n_i.\}$

Informally, $\text{cycle-extension}(\pi, x)$ is the set of def-clear paths with respect to x formed by following π , taking "side-trips" which traverse cycles zero or more times. For any du-path π with respect to x , $\pi \in \text{cycle-extension}(\pi, x)$.

Our modified version of all-du-paths requires that one element of the cycle-extension of each du-path be selected. We now define this criterion formally and investigate some of its properties.

DEFINITION: A test T which executes the set \mathcal{P} of paths satisfies the *cycle-extended-du-paths* criterion for a program P if and only if for each variable x and each du-path π with respect to x , \mathcal{P} covers some path $\pi' \in \text{cycle-extension}(\pi, x)$.

LEMMA: Let π be a def-clear path with respect to x from a node i containing a definition of x to a node j or edge (j, k) containing a use of x . Then there is a (not necessarily unique) du-path π' such that $\pi \in \text{cycle-extension}(\pi', x)$.

PROOF: Let $\pi = (n_1, n_2, \dots, n_k)$ be a definition-clear path with respect to x . The following algorithm decomposes π into the form $(\lambda_1, \lambda_2, \dots, \lambda_h)$ where for $1 \leq j \leq h \leq k$, λ_j begins and ends with n_{i_j} and $\pi' = (n_{i_1} \dots n_{i_h})$ is a du-path with respect to x . Thus $\pi \in \text{cycle-extension}(\pi', x)$.

begin

```
{initialize cycle-traversals to the empty path.}
for j := 1 to k do  $\lambda_j := ()$ ;
 $\pi' := ()$ ;
```

```
i := 1;
```

```
h := 0;
```

```
while i ≤ k do
```

```
begin
```

```
h := h + 1;
```

```
 $\pi' := \text{concat}(\pi', n_i)$ ;
```

```
{scan rest of path looking for last occurrence of  $n_i$ }
```

```
for j := i to k do
```

```
if  $n_i = n_j$  then last := j;
```

```

    if (i = 1) and (j = k) then last := 1; {special case}

    { $\lambda_h$  is subpath from  $n_i$  to  $n_{last}$ }
    for j := i to last do concat( $\lambda_h, n_j$ );

    {set i to position of first node in next subpath}
    i := last + 1;
end {while}
end;

```

The algorithm involves processing π from left to right, extracting maximal length subpaths which begin and end with the same node. The concatenation of the first nodes in these subpaths is the desired du-path. Some special care is required when π begins and ends with the same node. In this case, λ_1 is set to the subpath (n_1) and the du-path which is eventually produced is a simple cycle.

■

THEOREM 3: all-du-paths \Rightarrow cycle-extended-du-paths \Rightarrow all-uses

PROOF: The first inclusion follows immediately from the fact that every du-path belongs to its own cycle-extension. Suppose cycle-extended-du-paths \nRightarrow all-uses. There is a program P, a test T executing the set of paths \mathcal{P} through P which satisfies cycle-extended-du-paths, and some definition-use association which is not covered by \mathcal{P} . Assume that this association is a definition-c-use association, (i,j,x) . The proof for a definition-p-use association is nearly identical. Let π be a def-clear path with respect to x from i to j. By the lemma, there is a du-path π' with respect to x such that $\pi \in \text{cycle-extension}(\pi', x)$. Since T satisfies cycle-extended-du-paths, there is a path $\pi'' \in \text{cycle-extension}(\pi', x)$ such that \mathcal{P} covers π'' . π'' is a def-clear path with respect to x from i to j. So \mathcal{P} covers the association (i,j,x) . Contradiction.

It remains to show that the inclusions are strict. For any program P with no loops, a test satisfies all-du-paths for P if and only if it satisfies cycle-extended-du-paths for P. This fact, along with the proof in [RAP85] that all-du-paths strictly includes all-uses (which involved a loop-free program) shows that cycle-extended-du-paths strictly includes all-uses.

To see that all-du-paths strictly includes cycle-extended-du-paths, consider the program in Figure 9. The test {2} which executes $\mathcal{P} = \{(1,2,3,4,3,4,3,5,6)\}$ satisfies cycle-extended-du-paths but does not cover the du-path $(2,3,5)$. ■

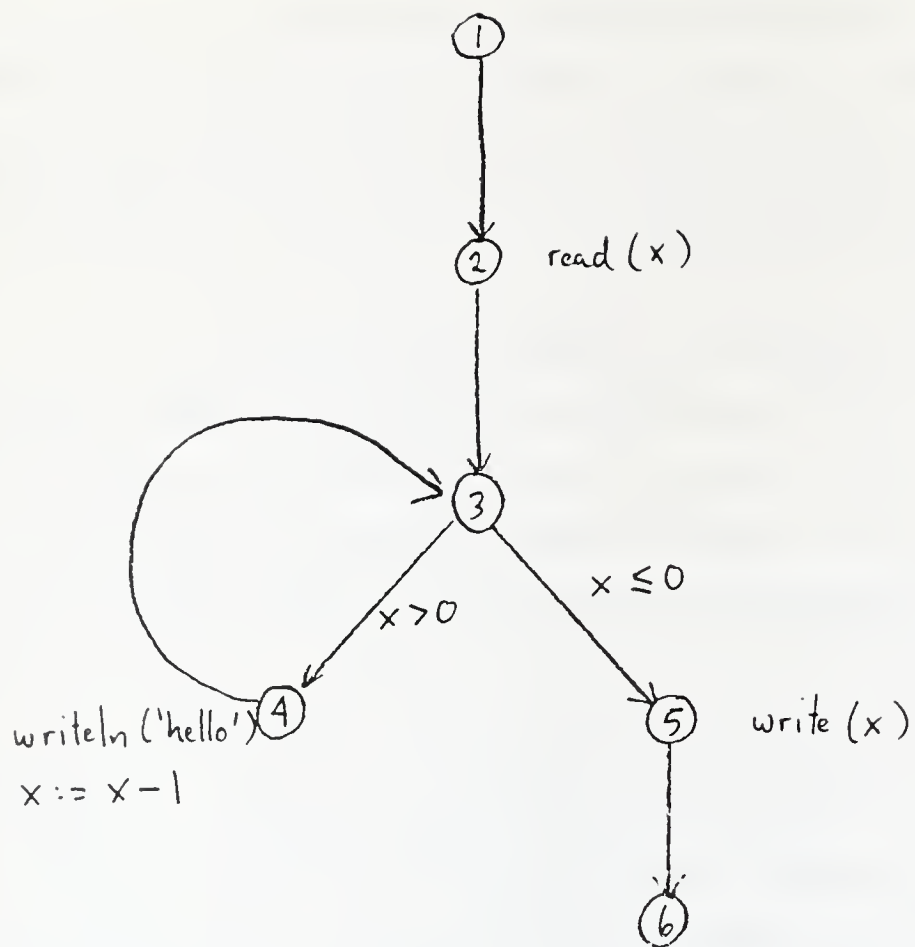


FIGURE 9

We next investigate the criterion (cycle-extended-du-paths)*.

DEFINITION: A test T which executes the set \mathcal{P} of paths satisfies the (cycle-extended-du-paths)* criterion for a program P if and only if for each variable x and each du-path π with respect to x for which $\text{cycle-extension}(\pi, x)$ contains at least one executable path, \mathcal{P} covers some path $\pi' \in \text{cycle-extension}(\pi, x)$.

THEOREM 4: $(\text{all-paths})^* \Rightarrow (\text{cycle-extended-du-paths})^* \Rightarrow (\text{all-uses})^*$

PROOF: It is obvious that $(\text{all-paths})^*$ includes $(\text{cycle-extended-du-paths})^*$. The proof that $(\text{cycle-extended-du-paths})^* \Rightarrow (\text{all-uses})^*$ is similar to the unstarred case. Suppose $(\text{cycle-extended-du-paths})^* \not\Rightarrow (\text{all-uses})^*$. Then for some subprogram P and set \mathcal{P} of paths through P executed by a test which satisfies $(\text{cycle-extended-du-paths})^*$ there is an executable definition-use association which is not covered by \mathcal{P} . Assume that this association is a definition-c-use association, (i, j, x) . The proof for a definition-p-use association is nearly identical. Since (i, j, x) is executable, there is an executable definition-clear with respect to x path π from i to j . By the lemma, $\pi \in \text{cycle-extension}(\pi', x)$ for some du-path with respect to x , π' from i to j . Since the subset of $\text{cycle-extension}(\pi', x)$ consisting of executable paths is non-empty (it contains π) there is some executable path $\pi'' \in \text{cycle-extension}(\pi', x)$ which is covered by \mathcal{P} . π'' is a definition-clear path with respect to x from i to j so \mathcal{P} covers the association (i, j, x) .

The strictness of the inclusion follows from the proof of the strictness of $\text{cycle-extended-du-paths} \Rightarrow \text{all-uses}$ since the program in Figure 9 has no unexecutable paths. ■

5. Conclusions

We have introduced a new family of path selection criteria derived from the data flow testing criteria and explored the relationships among them. These criteria, the feasible data flow testing criteria, are obtained from the corresponding data flow testing criteria by eliminating unexecutable associations from consideration.

For a large class of "well-behaved" programs, the FDF criteria $(\text{all-p-uses})^*$, $(\text{all-p-uses/some-c-uses})^*$, and $(\text{all-uses})^*$ "bridge the gap" between $(\text{all-branches})^*$ and $(\text{all-paths})^*$ in the same way

that the corresponding DF criteria do. For certain programs with anomalies however, there are tests which satisfy $(\text{all-p-uses})^*$ without satisfying $(\text{all-edges})^*$.

Although $(\text{all-paths}) \Rightarrow (\text{all-du-paths}) \Rightarrow (\text{all-uses})$, $(\text{all-du-paths})^*$ does not even include $(\text{all-nodes})^*$. We have defined a new criterion, $(\text{cycle-extended-du-paths})$ such that $(\text{all-paths}) \Rightarrow (\text{cycle-extended-du-paths}) \Rightarrow (\text{all-uses})$ and $(\text{all-paths})^* \Rightarrow (\text{cycle-extended-du-paths})^* \Rightarrow (\text{all-uses})^*$.

The advantage of the FDF criteria over the DF criteria is that they satisfy the applicability property: for every subprogram P and every FDF criterion C there is some set of paths which is C-adequate for P. The DF criteria do not satisfy this property. The disadvantage of the FDF criteria is that it is undecidable whether a particular set of paths is C-adequate for P. Thus, in deciding whether to use the DF criteria or the FDF criteria, one is faced with a trade-off between applicability and automatability.

Although it is in general undecidable whether a given association is executable, it is often easy for a person looking at a subprogram to determine whether or not a particular association is executable. Sometimes this requires very little semantic information. For example, any program with a for loop in which the upper bound is always greater than or equal to the lower bound has an unexecutable definition-p-use association. In other cases, determining whether a given association is executable seems to require "high-level" understanding of how the subprogram and other subprograms which it calls operate.

We plan to enhance ASSET, our data flow testing tool, by adding heuristics which attempt to determine which of the required associations are unexecutable. When the heuristics cannot decide whether or not a particular association is executable, the person using the tool will have to intervene. We hope that this approach will prove to be a practical way to preserve the applicability property enjoyed by the FDF criteria while sacrificing automatability to as small extent as possible.

6. References

- [CLA85] L.A. Clarke, A. Podgurski, D.J. Richardson and S.J. Zeil, "A Comparison of Data Flow Path Selection Criteria", *8th IEEE International Conference on Software Engineering*, August, 1985, London, pp. 244-251.
- [FRA85a] P.G. Frankl, S.N. Weiss and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.
- [FRA85b] P.G. Frankl and E.J. Weyuker, "A Data Flow Testing Tool," *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.
- [GIR85] M.R. Girgis and M.R. Woodward, "An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis," *8th IEEE International Conference on Software Engineering*, August, 1985, London, pp. 313-319.
- [GOO75] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, SE-1,2, June 1975.
- [HEC77] M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, 1977.
- [HOW76] W.E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2,3, 1976, pp. 208-215.
- [HOW78] W.E. Howden, "A Survey of Dynamic Analysis Methods" in *Tutorial: Software Testing and Validation Techniques*, eds. E. Miller and W.E. Howden, IEEE Computer Society, 1978.
- [HUA75] J.C. Huang, "An Approach to Program Testing," *ACM Computing Surveys*, 7(3), Sept. 1975, pp. 113-128.
- [KOR85] B. Korel and J. Laski, "A tool for Data Flow Oriented Program Testing", *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.
- [LAS83] J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Trans. Software Eng.*, Vol. SE-9, No.3, May 1983, pp. 347-354.
- [MIL74] E. F. Miller, Jr. M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation," *Dig. Compcon*, Spring 1974, pp.161-164.
- [NTA84] S. Ntafos, "On Required Element Testing," *IEEE Trans. Software Eng.*, Vol. SE-10, No.6, Nov. 1984, pp. 795-803.
- [OST76] L.J. Osterweil and L. D. Fosdick, "DAVE -- A Validation Error Detection and Documentation System for Fortran Programs," *Software Practice and Experience*, Oct-Dec 1976, pp. 473-486.
- [RAP82] S. Rapps and E. J. Weyuker, "Data Flow Analysis Techniques for Program Test Data Selection," *Proceedings Sixth Int'l Conference on Software Eng.*, Tokyo, Japan, Sept. 1982, pp. 272-278.
- [RAP85] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, No.4, April 1985, pp. 367-375.
- [SCH73] M. Schaeffer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N. J., 1973.
- [WEY85] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy," Computer Science Department Technical Report #99, Courant Institute of Mathematical Sciences, New York, NY, Jan. 1984, revised Aug. 1985.
- [WOO80] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with Path Analysis and Testing of Programs," *IEEE Trans. Software Eng.*, Vol. SE-6, May 1980, pp. 278-286.

APR 11 1986

A fine will be charged for each day the book is kept overtime.

GAYLORD 142			PRINTED IN U.S.A.

NYU COMPSCI TR-208
Frankl, Phyllis G
Data flow testing in the
presence of unexecutable
paths. c.2

LIBRARY
N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012

